Enabling High-Level Application Development in Internet of Things

Pankesh Patel

Enabling High-Level Application Development in Internet of Things

Pankesh Patel
INRIA Paris-Rocquencourt, France
pankesh.patel@inria.fr

September 10, 2012

Abstract

The Internet of Things (IoT) combines Wireless Sensor and Actuation Networks (WSANs), Pervasive computing, and the elements of the "traditional" Internet such as Web and database servers. This leads to the dual challenges of scale and heterogeneity in these systems, which comprise a large number of devices of different characteristics. In view of the above, developing IoT applications is challenging because it involves dealing with a wide range of related issues, such as lack of separation of concerns, need for domain experts to write low level code, and lack of specialized domain specific languages (DSLs). Existing software engineering approaches only cover a limited subset of the above-mentioned challenges.

In this work, we propose an application development process for the IoT that aims to comprehensively address the above challenges. We first present the semantic model of the IoT, based on which we identify the roles of the various stakeholders in the development process, *viz.*, domain expert, software designer, application developer, device developer, and network manager, along with their skills and responsibilities. To aid them in their tasks, we propose a model-driven development approach which uses customized languages for each stage of the development process: *Srijan Vocabulary Language* (SVL) for specifying the domain vocabulary, *Srijan Architecture Language* (SAL) for specifying the architecture of the application, and *Srijan Network Language* (SNL) for expressing the properties of the network on which the application will execute; each customized to the skill level and area of expertise of the relevant stakeholder. For the application developer specifying the internal details of each software component, we propose the use of a customized generated framework using a language such as Java. Our DSL-based approach is supported by code generation and task-mapping techniques in an application development tool developed by us. Our initial evaluation based on two realistic scenarios shows that the use of our techniques/framework succeeds in improving productivity while developing IoT applications.

1 Introduction

The IoT has been discussed in literature for some time now, albeit with several similar but non-identical definitions. In this paper, we build upon the following definition, proposed by the CASAGRAS project [4] in 2009:

> "A global network infrastructure, linking physical and virtual objects through the exploitation of data capture and communication capabilities. This infrastructure includes existing and evolving Internet and network developments. It will offer specific object-identification, sensor and connection capability as the basis for the development of independent cooperative services and applications. These will be characterized by a high degree of autonomous data capture, event transfer, network connectivity and interoperability."

The IoT has recently moved closer to being a reality, thanks to the increased abundance of smart phones, WSANs [1], Web Services, and RFID technologies [33]. Several IoT applications have been reported in recent research, and we expect to see increased adoption of IoT concepts in the fields of personal health, inventory management, and domestic energy usage monitoring [2]. It is noteworthy that the systems in IoT include both WSAN devices as well as smart appliances, in addition to the elements of the "traditional" Internet such as Web and database servers; these can be employed to design complex applications, such as the one discussed next.

1.1 Illustrative Application: Smart Office

We consider the following office environment management application to illustrate the characteristics of IoT applications. An office might consist of several buildings, with each building in turn consisting of one or more floors, each with several rooms, each instrumented with a large number of heterogeneous devices with

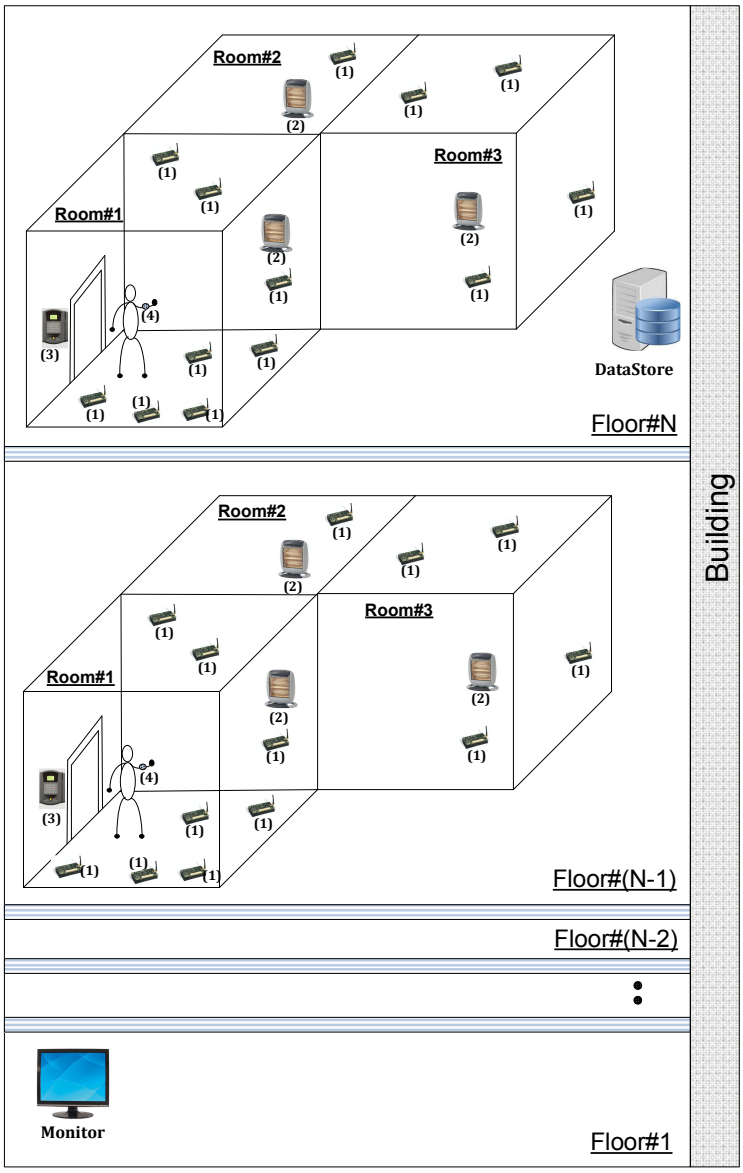

Figure 1: Multi-floored building with deployed devices ((1)-device with temperature sensor, (2)-Heater, (3)-BadgeReader, (4)-Badge)

sensors, actuators, and storage (see Figure 1). The system aims to regulate appropriate temperature for worker productivity and personal happiness, and also provides general information such as current temperature of the building.

The temperature in each room of the building is regulated by a sense-compute-actuate executing among the temperature sensors and heaters of the room, using the average temperature of room. Additionally, average temperature values are computed at the per-floor and per-building levels to be displayed at monitors at the building entrance and at the control station. When a user enters or leaves a room, a badge reader detects this event, and queries a central employee database for the user's preferences. Based on these, the threshold used by the room's devices is updated.

1.2 Challenges and Contributions

An important challenge to be addressed in the IoT is *ease of application development*. There is a growing awareness about this problem in the research community, and an increasing number of approaches [6, 8, 19, 23, 25, 27] are being proposed in the closely related fields of WSANs and Pervasive computing (discussed in more detail in the Section 5). While the main challenge in the former is the extremely *large scale* of the systems (hundreds to thousands of largely similar nodes), the primary concern in the latter has been the *heterogeneity* of devices. IoT applications such as the one discussed in the Section 1.1 raise the following challenges in the application development process:

- **No separation of concerns.** IoT applications incorporate knowledge in a number of fields such as the application domain, distributed systems, embedded system, networking, and operating systems. To develop such applications, developers require multiple skills such as dealing with domain specific expertise, low-level hardware knowledge, design of distributed code, lots of administrative code to interface hardware and software components. Nearly all the application development approaches proposed for IoT assume only *one* role – the programmer/developer – to be played by the individuals developing the application. This hinders rapid application development, and is unsuited to developing sophisticated large scale applications.

- **"General-Purpose" DSLs.** While there are some approaches based on metamodels/domain-specific languages [8, 25, 27] for the IoT, domain specific development approaches stop short at providing a "one-size fits all" meta-model or DSL for the entire IoT domain. We believe that it is too large a granularity, and an ideal approach would provide Domain Specific Software Engineering (DSSE) [31] specific to the application domain, e.g., HVAC, in order to maximize the ease of use and other benefits obtained from abstractions.

- **Heterogeneity of target devices.** IoT systems are expected to be extremely heterogeneous - both in terms of the capabilities of the devices involved, as well as in terms of the different implementations of the sensors/actuators to be used (e.g., several vendors might sell temperature sensors, but with slightly different capabilities). Devices feature various interaction modes (e.g., command, request-response, publish-subscribe). This heterogeneity spreads in the application code, cluttering it with low-level detail. It should not be the developer's responsibility to handle this heterogeneity, since ideally the *same* application should execute on entirely different deployments (e.g. the same smart building application on different offices).

- **Scale.** Applications in the IoT may execute on systems consisting of thousands of devices. Requiring the ability of reasoning at such levels of scale is impractical, even for computer science experts, let alone domain experts. Consequently, there is the need for adequate abstractions to present the large scale in a suitable manner to the stakeholders involved.

Based on above identified characteristics of IoT application, our **research goal is**:

> *"Enable high-level application development that allows stakeholders involved in the development of IoT applications to easily specify applications, which involve a large number of heterogeneous devices."*

Our work aims to address the above problems by making the following contributions[1]:

- **Semantic Model for the Internet of Things.** In order to achieve clarity on the task at hand while developing IoT applications, we first clearly identify the concepts of IoT and relationships among them. More details of our semantic model are in Section 2.

- **Identification of Roles in the Development Process.** Leveraging the semantic model above, we identify the precise role of each stakeholders involved in the development of IoT applications. This promotes separation of concerns and raises the level of knowledge that can be shared by other stakeholders. This clear identification of the expectations and specialized skills of each type of stakeholder is an integral part in our work on supporting each of them in playing their part effectively to smoothen the application development process.

- **A multi-stage approach for IoT application development.** In order to support the different stages (i.e. design, implementation, and deployment) of the IoT application development process resulting from

[1]This work does not claim to completely solve the IoT research challenges discussed above. Our main focus is to present a subset of tasks involved in the IoT application development process, and the interrelationships among them. By clearly identifying the tasks, we can help researchers in the community attack the particular subtasks involved in process.

the actions of each of the stakeholders, we propose a multi-stage model-based approach. For each type of stakeholder, we provide a separate set of abstractions, customized not only for the IoT in general, but also for the application domain itself (*e.g.*, HVAC). This allows the stakeholder to specify their share at the *proper* level of abstraction commiserate with the skills and the responsibilities of the role he is playing. The details of the various stakeholder roles introduced above, as well as this process is discussed in more detail in Section 3.

- **A generative approach of application development.** We build an application development tool, combined with our language, provide support to various stakeholders to design and develop applications at each stages of IoT application development. The current version of our tool includes two code generators. First, *a*n application code generator produces skeleton classes from high-level specification (expressed using DSL at design stage). Second, *a* deployment code generator produces device-specific code to result in a distributed software system collaboratively hosted by individual devices in the IoT. More details of our tool are in Section 3.3; we further show by our experiments (discussed in Section 4) that using our framework, the application development burden is significantly reduced.

In addition to the above, we present the related work in this area in Section 5. We summarize our contribution so far in Section 6 and conclude with our plan for future work in Section 7.

2 Semantic Model for the Internet of Things

We have extracted the concepts and associations that we believe are suitable for representing applications in the Internet of Things [22]. We present the semantic model derived from them in the section below. A graphical representation of the same is in Figure 2, which follows the notation introduced in [17]. The figure contains the concepts in the IoT domain, along with their relationships with each other, including the cardinality of such relationships.

2.1 Concepts

The concepts in the Internet of Things can be divided broadly into two categories:

2.1.1 "Traditional" Internet concepts

These are concepts which developers of Internet applications are most familiar with, namely:

A **software component** (using the definition from [30]) is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context. We identify the following types of software components:

- A **computational service** is a software component that takes one or more units of information as input and produces an output. It is a representation of the *processing* taking place in the application.
- A **storage service** provides read and write access to persistent data. This data can be accessed by other software entities by interacting with the storage service.
- An **end-user application** is a software component that is designed to help a user to perform tasks by interacting with other software components. For instance, in the room temperature maintenance application, a user can provide his temperature preferences to the IoT application using an app installed on his smart phone.

Some concepts related to the ones above are that of a **user**, which is a human who performs singular or multiple related specific tasks; a **store**, which is the entity that actually hosts the data (in the above example, the various databases managed by the MySQL server are instances of store); and **information**, which is any data that is meaningful by itself.

2.1.2 "Thing"-oriented concepts

In addition to the ones above, our semantic model also includes the following concepts, which are specifically used to model the "things" in the IoT.

- An **entity of interest (EoI)** [11] is a real world object, including the attributes that describe it, and its state that is relevant from a user or an application perspective. For instance, the EoI may be any real world objects such as room, book, plant, etc.

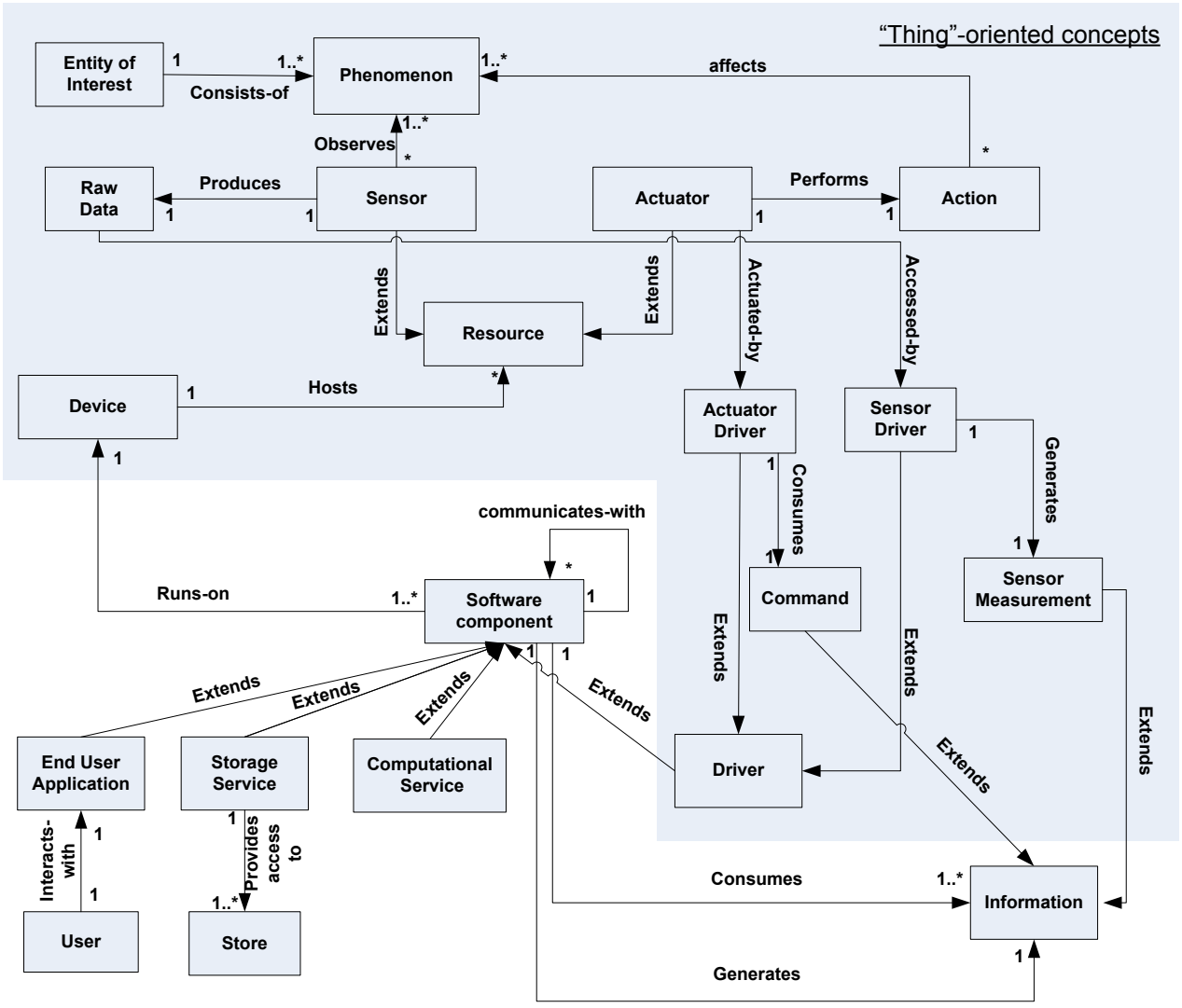

Figure 2: Semantic Model

- A **phenomenon** [11] a property of a physical entity that is observable. For instance, the temperature of a room and ID of tag are the examples of the phenomenon.
- A **resource** [11] is a conceptual representation of a sensor or an actuator, where a **sensor** is a type of resource that has the ability to detect changes in the environment. Thermometer, and tag readers are examples of sensors. An **actuator** is a resource that has the ability to make changes in the environment through an action. Heating or cooling elements, speakers, lights, etc. are examples of actuators.
- **Raw data** is a representation of a sensor observation. For instance, the raw data reading generated by a temperature sensor can be the number 25, without any explicit meaning or units attached to it. Note that this is different from *information* described above, which attaches additional data such as unit of measurement (Celsius/Fahrenheit). Thus, "25° Celsius" is the information.
- An **action** represents the act of an entity in the environment. For instance, "switching the lights on".
- A **device** [12] is an entity that provides sensor and actuator resources the ability to communicate with other entities. Without a device, resources can not interact with other resources. Tag readers, mobile phones, and personal computers (PC) are all example devices.
- A **sensor driver** accesses raw data and further describes it by attaching meta-data such as unit of measurement, time of sensing, etc. with raw data. The result is called **sensor measurement**, which is a type of information.
- A **command** is an instruction that describes a desired outcome. For instance, "switch the heater on", "turn on the light". An **actuator driver** translates a command and triggers the actuator appropriately.

For instance, the heater driver translates a command (such as "switch the heater on") and turns the heater on as a result. Note that a both sensor and actuator drivers are types of **driver**, which in turn is a type of software component.

2.2 Associations

The associations between the concepts in the Internet of Things are as follows:

- An EoI **consists-of** one or more phenomenon.
- A sensor **observes** a phenomenon and **produces** data about it.
- An actuator **performs** an action that **affects** a phenomenon.
- A device **hosts** zero or more resources. E.g., a smart phone might host resources such as an accelerometer sensor, a light sensor, etc.
- A software entity **runs-on** a device. E.g., an instance of the Jetty tiny web server running on a home desktop.
- Raw data is **accessed by** a sensor driver that **generates** a sensor measurement. E.g., in temperature monitoring, the temperature raw data is accessed by the temperature sensor that generates the temperature measurement.
- An actuator is **actuated by** an actuator driver that **consumes** an actuation command.
- A user **interacts-with** an end-user application for various purposes such as setting the temperature preference, hearing an alert message, etc.
- A storage service **provides access to** a store.
- Software components **communicate-with** each other to exchange data and control, in a manner similar to that described under the functions of a software connector in [30], which might contain instances of various interaction paradigms [10] such as message passing, publish-subscribe etc.

3 Multi-stage Model-driven Approach for IoT Application Development

The application development process involves several stakeholders, each playing his specialized role. We believe that there is a lack of software engineering techniques for IoT application that adequately empower each type of stakeholder while requiring them to only specify those parts of the application that they are experts in.

Taking inspiration from 4+1 view model of software architecture [16] and tool-based methodology for pervasive computing [7], we propose to divide the responsibilities of the stakeholders in the IoT application development process into five distinct roles, whose skills and responsibilities are stated in Table 1.

Based on the roles defined in Table 1, we propose the following multi-stage development process, detailed in Figure 3, consisting of the following steps:

1. **Domain Vocabulary Specification.** We provide the *domain expert* with a vocabulary language (discussed in Section 3.1) to specify vocabulary of an application domain (e.g., HVAC)(step 1 in Figure 3). The vocabulary includes the specification of entities (i.e, sensor, actuator, and storage service) that are responsible for interacting with a phenomenon of an entity of interest; it also includes the definition of partitions that the system is divided into.
2. **Application Architecture Specification.** Given the vocabulary, we provide the *software designer* with an architecture language (discussed in Section 3.2) to specify an architecture of an application (step 2 in Figure 3), specifying the details of the computational, controller, as well as how they communicate-with other software components.
3. **Implementing Application Logic.** Our approach leverages both vocabulary and architecture specification to provide support to *application developer*. To describe the internal logic of each software component, application developer is provided a customized programming framework, pre-configured according to the architecture of an application, an approach similar to the one discussed in [6] (step 3 in Figure 3). These consist of abstract classes[2] for each software component that hide low-level communication details, and allow the developer to focus only on the code that will implement the logic of that software component (step 4 in Figure 3).

[2]We assume that the developer uses an object-oriented language.

Role	Skills	Responsibilities
Domain Expert [32]	Understands domain concepts, including the data types produce by the sensors and consumed by actuators, as well as how the system is divided into partitions.	Specify the vocabulary for all applications to be used in the domain.
Software Designer [32]	Software architecture concepts, including the proper use of interaction pattern such as publish-subscribe and request-response for use in the domain.	Define the structure of the application by specifying the software components and their producer/consumer relationships.
Application Developer [6]	Skilled in algorithm design and object-oriented programming languages.	Develop the internal code for the computational services, and controllers in the application.
Device Developer	Deep understanding of the inputs/outputs, and protocols of the individual devices.	Write drivers for the sensors and actuators used in the domain.
Network Manager	Deep understanding of the specific target area where the application is to be deployed.	Install the application on the system at hand; this process may involve the generation of binaries or bytecode, and configuring middleware.

Table 1: Roles in the IoT Application Development Process

4. **Target Network Specification.** We provide a *network manager* with the network language (discussed in Section 3.4) to specify target deployment scenario. For each target deployment scenario where the application is to be deployed, the respective network manager specifies (step 5 in Figure 3) the details of the devices in his system, using the concepts defined in the vocabulary.

5. **Generating deployment code.** To generate deployment code, this stage consists of two core sub-stages: *Mapper* and *System Linker*.

   - **Mapper.** The mapper outputs a mapping of set of instantiated software components to the set of devices (step 6 in Figure 3). The mapper takes inputs as set of *instantiation rules* of software components from architecture specification and *set of devices* from network description. The mapper decides the specific device where each software components will be running [3].
   - **System Linker.** This module combines the information generated by the various stages of the compilation into the actual code to be deployed on the real devices (step 8 in Figure 3). It merges the generated code, the imperative code provided by the *application developer*, the generated code from mapper module, and device specific drivers provided by *device developer* The output of this module is a deployment code, a set of Java packages for each device. These packages are not executable. They still need to be compiled by device-level compiler designed for the target platform.

Based on our analysis of the roles played by the various stakeholders in the stages described above, we have designed DSLs, each named after *Srijan*, the Sanskrit word for "creation". We provide the details of above mentioned stages with DSL, using the application introduced in Section 1.1 to show our approach can be used by various stakeholders in the stages of the application development process.

For reader's clarity, we illustrate a layered architecture of a *Smart Office* application in Figure 4 (similar to one discussed in [6]). The bottom part of the figure shows *BadgeReader* to identify a Badge that is entering a room, a *ProfileDB* to store the association between Badge and its temperature preference, temperature sensor to sense temperature of room. A *proximity* service coordinates event from BadgeReader with the contents of the ProfileDB. This information is passed to the *RegulateTemp* controller, which takes decision that are carried out by invoking *Heater* actions (top of the Figure 4). Additionally, this application aims to display current temperature of multi-floored building using screen placed at entrance of the building. The temperature data is first routed to a local average temperature service (i.e. *RoomAvgTemp*), deployed in per room, then later per floor (i.e., *FloorAvgTemp*), and then ultimately routed to building central average temperature service (i.e., *BuildingAvgTemp*). Finally, the calculated value is delivered to controller (i.e., *ManageTemp*), which triggers a display action of *Monitor* (top of the Figure 4).

[3] Current version of mapper maps software component to device randomly.

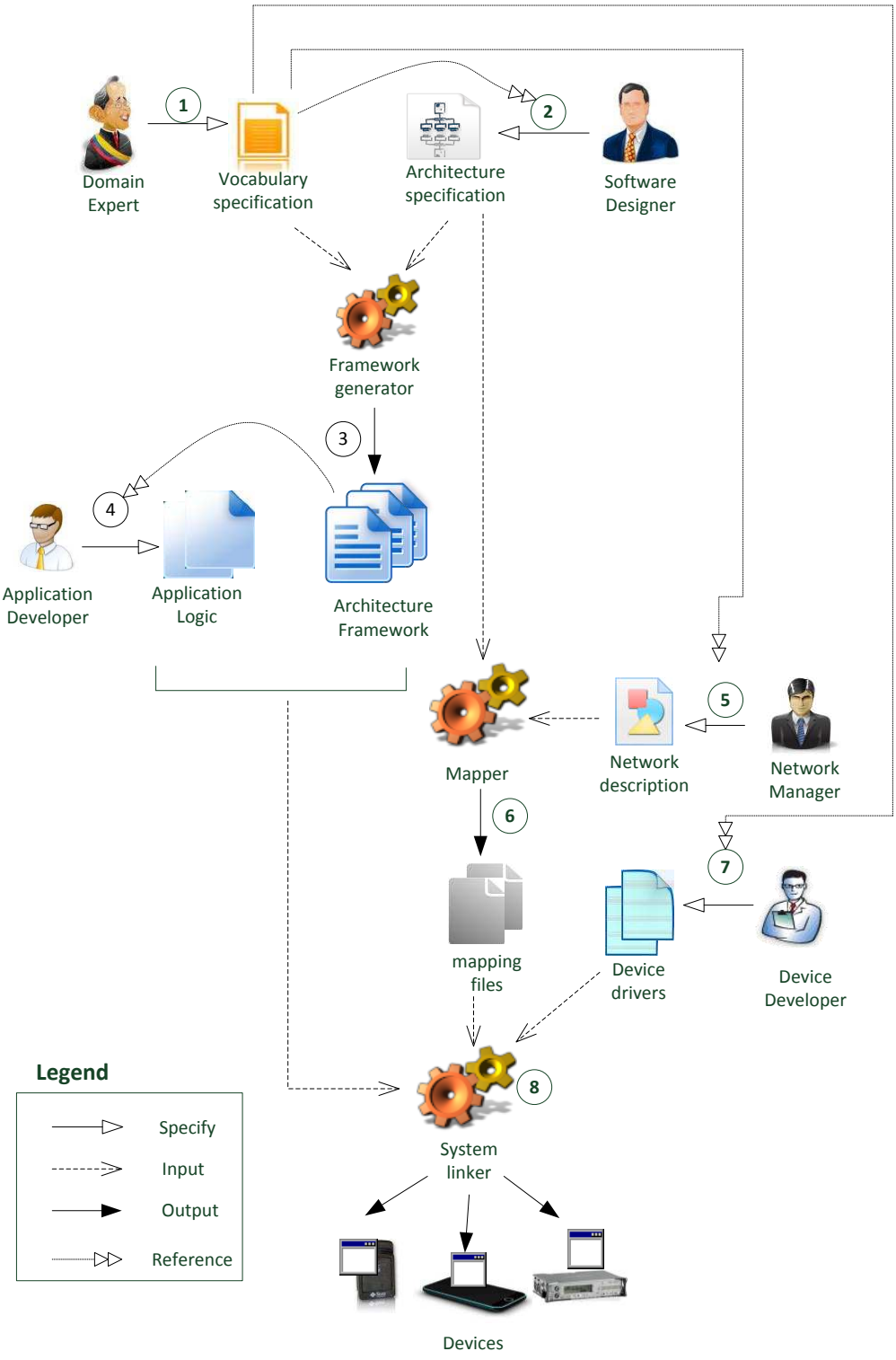

Figure 3: Development Cycle

3.1 Domain Vocabulary Specification

The Srijan Vocabulary Language (SVL) (for grammar, see Appendix A.1) is designed to enable the *domain expert* to describe the vocabulary of the domain; namely the *abilities* each node might possess — types of sensors, actuators, or storages they are connected to along with the input/output data types of each — and the *regions* that the target deployment area can be divided into. The language offers following domain constructs to interact with phenomenon of an EoI that maps into the concepts that are relavant to IoT :

- *abilities:* These define the properties that will be used to characterize individual devices. They include, for sensors, the sensor measurements produced by them; for actuators, the command consumed by them;

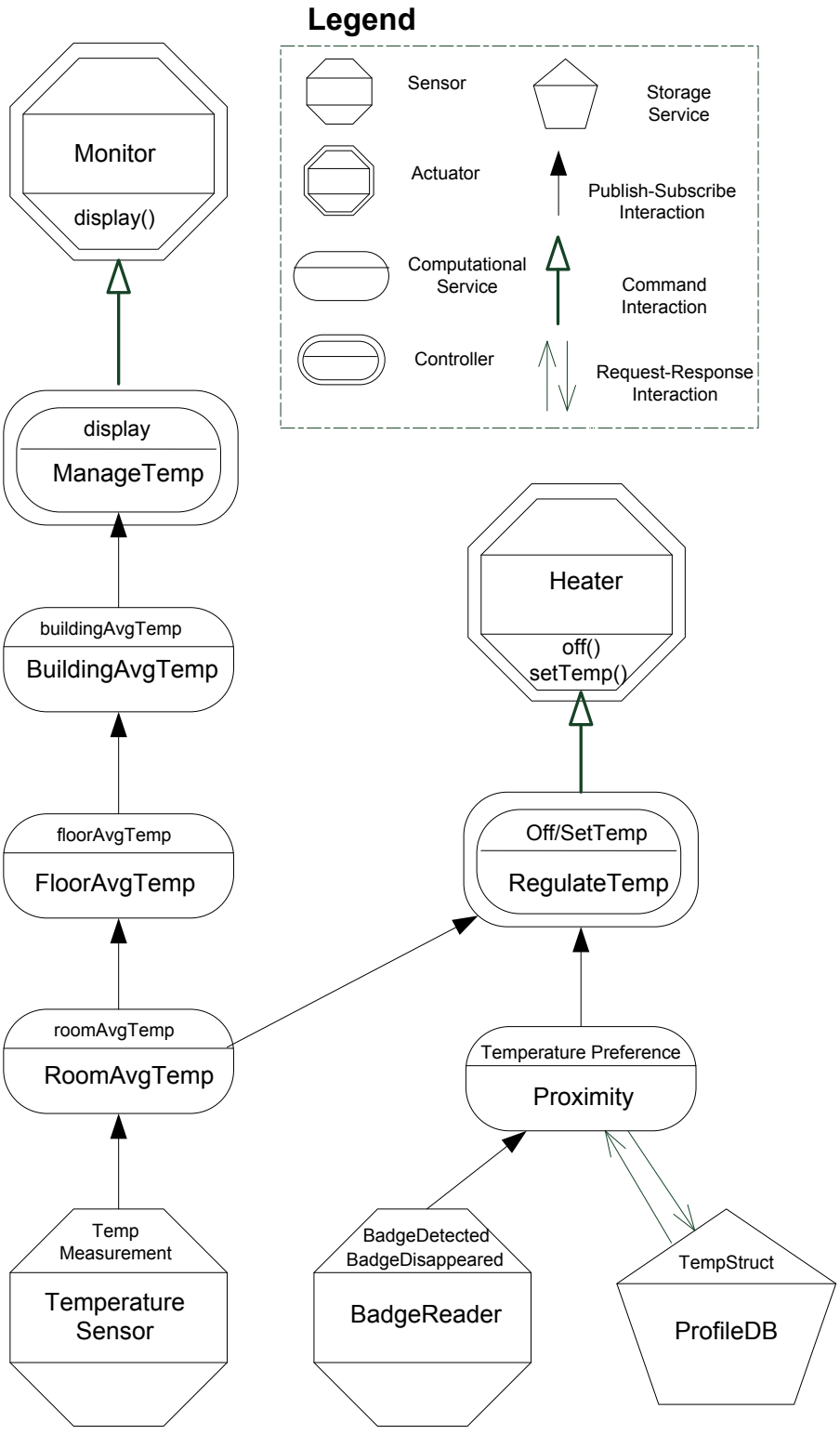

Figure 4: Layered architecture of the smart office application in Section 1.1

and for databases, the type of data that can be accessed.

- *regions:* These define the various region labels that can be used to define the (logical) locations of the devices and scopes from which the software components will produce/consume data. *e.g.*, HVAC applications reason in terms of room and floors, while smart city applications can be best expressed in terms of city blocks.

We present the SVL by examing a code snippets of vocabulary specification of smart office application (see Listing 1). Each sensor and its ability is declared using `sensors` and `generate` keyword respectively. For example, `BadgeReader` defines its two sensor measurements `badgeDetected`, and `badgeDisappeared` (see Listing 1, line 20-23). The sensor measurement of `BadgeReader` are defined using two structures (see Listing 1, line 7-13): `BadgeDetectedStruct`, and `BadgeDisappearedStruct`.

Actuator and its actions are declared using `actuators` and `action` keywords respectively. Some action of actuator takes one or more inputs, specified as *parameters* of an action. For example, `Heater` declaration defines `Off()`, and `SetTemp(setTemp:TempStruct)` actions (see Listing 1, line 26-28).

Storage is declared using `storages` keyword (see Listing 1, line 31-32). Retrieval of data from storage service requires a *parameter*. For example, `profile` data access of `ProfileDB` storage service maps a `badgeID` to `TempStruct`; in this case, the data access need to be parametrized by a badge identifier. Such a parameter is introduced by `accessed-by` keyword.

To enable the definition of system partition at logical level, we provide domain expert with *Region* construct seen in programming languages for WSAN such as Regiment [21], Hood [34], TAG [18], and ATaG [23]. It is declared using `regions` keyword. This construct divides the system into hierarchical regions (e.g., room, floor, building) with its type (e.g., String, Integer)[4], thus defining region as `regionName:regionType` (see Listing 1, line 1-4).

```
1  regions:
2    Building : Integer;
3    Floor : Integer ;
4    Room : Integer;
5
6  structs :
7    BadgeDetectedStruct
8      badgeID : String;
9      timeStamp : long;
10
11   BadgeDisappearedStruct
12     badgeID : String;
13     timeStamp : long;
14
15   TempStruct
16     tempValue : double;
17     unitOfMeasurement: String;
18
19 abilities:
20   sensors:
21     BadgeReader
22       generate badgeDetected : BadgeDetectedStruct;
23       generate badgeDisappeared :  BadgeDisappearedStruct;
24
25   actuators:
26     Heater
27       action  Off();
28       action SetTemp(setTemp :TempStruct);
29
30   storages:
31     ProfileDB
32       generate  profile : TempStruct  accessed-by badgeID : String;
```

Listing 1: Code snippets of smart office application architecture. Keywords are printed in ***word***.

[4]Current version of our work supports only `Integer` datatype.

We now present Srijan Architecture Language (SAL). Note that the region labels and data structures defined using SVL in the vocabulary become keywords in the SAL customized for the domain. This can, in turn, be exploited by tools to give code completion help to the software designer.

3.2 Application Architecture Specification

Given a vocabulary, SAL (for grammar, see Appendix A.2) is designed to enable the *software designer* to design an application. It incorporates Sense-Compute-Control (SCC) [5], an architecture pattern commonly used in the pervasive computing application. This architecture pattern consists of *computational services* fueled by sensors and storage. It computes gathered data to make it complainant to the application needs. The *controller* takes data from computational services as input and takes decision that are carried out by invoking the actuator. Following this architecture pattern, SAL contains the constructs to specify the following types of data-flow between each type of application-specific software component:

- For each controller, the data types it consumes and the actuator drivers to which it issues commands, labeled with scopes from the region labels defined in the vocabulary.
- For each computational service, the data types it consumes and produces, labeled again with the scopes from which they should be gathered, as well as data to be accessed from storage services.

We present the SAL by examining a code snippets of smart office application (see Listing 2). This code snippets is devoted to `Proximity` component (see Listing 2, line 9-14), which coordinates events from the `BadgeReader` with the content of `ProfileDB` storage service. To do so, `Proximity` processes three sources of information: one for entering badge (i.e., badge detection), one for leaving badge (i.e., badge disappeared) and one for requesting user's temperature profile from `ProfileDB` that is expressed using `request` keyword (see Listing 2, line 13). The former two source of information is declared using `consume` keyword that takes source name and data interest of component from logical region (see Listing 2, line 11-12). The declaration of `hops:0:room` indicates that the component is interested in consuming badge events of the current room. The `Proximity` component is in charge of managing badge events of *room*. Therefore, we need `Proximity` service to be partitioned per room using `in-region:room` (see Listing 2, line 14).

The output of the `Proximity` and `RoomAvgTemp` are consumed to the `RegulateTemp` component, declared using the `controller` keyword (see Listing 2, line 21-27). This component is responsible for taking decisions that are carried out by invoking command, declared using the `command` keyword (see Listing 2, line 25-26).

```
1  structs:
2    UserTempPrefStruct
3      tempValue : double;
4      unitOfMeasurement : String;
5      timeStamp : long;
6
7  softwarecomponents:
8    computationalService:
9      Proximity
10       generate tempPref : UserTempPrefStruct;
11       consume badgeDetected from hops : 0: Room;
12       consume badgeDisappeared from hops : 0 : Room;
13       request profile;
14       in-region: Room;
15
16   RoomAvgTemp
17       generate roomAvgTempMeasurement : TempStruct;
18       consume tempMeasurement from hops: 0 : Room ;
19       in-region : Room;
20
21     controller:
22       RegulateTemp
23         consume roomAvgTempMeasurement from hops : 0 : Room;
24         consume tempPref from hops : 0 : Room;
25         command Off() to hops : 0 : Room;
26         command SetTemp(setTemp) to hops : 0 : Room;
27         in-region : Room;
```

Listing 2: Code snippets of smart office application architecture. Keywords from vocabulary are printed in word. Language keywords are printed in ***word***.

3.3 Implementing Application Logic

This section presents an application development tool that leverages both vocabulary definition and achitecture specification and generate Java programming framework, thus maximizing the productivity of application developer and device developer. The current version of tool is composed of ANTLR[5] based parser and StringTemplate[6] based code generator. We now present a generated programming framework of smart office application.

3.3.1 Generated Programming Framework

A generated Java programming framework contains *abstract classes* corresponding to both vocabulary definition and architecture specification. The generated abstract classes also include abstract methods declarations to allow the *application developer* to program the application logic. The application developer implements each abstract method of generated abstract class in the subclass of abstract class. Moreover, the generated programming framework includes concrete methods that allow application developer to interact (i.e., publish/subscribe, request/response, command) with other components transparently, without dealing with the interaction details. We now present the generated code of each part of declaration in vocabulary and architecture specification.

A sensor publishes sensor measurements for computational services. Support for this propagation is generated by our framework generator in form of concrete methods, allowing sensor to fuel the computational services. For example, from the `BadgeReader` declaration in vocabulary specification (Listing 1, line 21-23), the implementation of both `setbadgeDisappeared()` and `setbadgeDetected()` methods (Listing 3, line 12-19) are generated in the abstract class (Listing 3, line 1-20). These methods are fueled by `BadgeReader` to propagate the `BadgeDetected` and `BadgeDisappeared` event in implementation of abstract methods `handlebadgeDisappeared()` and `handlebadgeDetected()` (Listing 3, line 8-10).

```
1  public abstract class BadgeReader {
2
3    private BadgeDisappearedStruct badgeDisappeared;
4
5    private BadgeDetectedStruct badgeDetected;
6
7    // This abstract methods are implemented by application developers to
        propogate events.
8    public abstract void handlebadgeDisappeared();
9
10   public abstract void handlebadgeDetected();
11
12   protected void setbadgeDisappeared(BadgeDisappearedStruct newValue) {
13       // publish "badgeDisappeared" event.
14       PubSubMiddleware.publish("badgeDisappeared", newValue);
15   }
16   protected void setbadgeDetected(BadgeDetectedStruct newValue) {
17       // publish "badgeDetected" event.
18       PubSubMiddleware.publish("badgeDetected", newValue);
19   }
20 }
```

Listing 3: The Java abstract class `BadgeReader` generated by framework generator from the declaration of the BadgeReader entity in vocabulary.

A computational service processes input data to produces refined data to its consumers. The input data is either published by entities (i.e., sensors, computational service) or requested by the computational service. The code for both interaction modes is produced by the framework generator. For example, from the `Proximity` declaration in architecture specification (Listing 2, line 11-12), the implementation of `notifyReceived()` method is generated to receive published events in the abstract class (Listing 4, line 2-9). Additionally, the framework generator produces the implementation of `getProfile()` method to request data from `ProfileDB` component (Listing 4, line 10-13). This method is called by the application developer in the subclass of `Proximity` to implement the application logic.

```
1  public abstract class Proximity {
2    public void notifyReceived(String eventName, Object arg) {
```

[5] http://www.antlr.org/
[6] http://stringtemplate.org

```
    if (eventName.equals("badgeDisappeared")) {
      // Do something, when badge disappeares.
    }
    if (eventName.equals("badgeDetected")) {
      // Do something, when badge detected.
    }
  }
  protected UserProfile getProfile(String badgeID) {
  // get user's temperature preference by sending command;
  return (UserProfile) PubSubMiddleware.sendCommand("getProfile", badgeID);
  }
}
```

Listing 4: The Java abstract class `Proximity` generated by framework generator from the declaration of the computational service `Proximity` in architecture.

A storage service provides read and write access of persistent data store. The framework generates concrete methods for both reading and writing data in the storage. For example, from the `ProfileDB` declaration in Vocabulary (Listing 1, line 31-32), `setProfile()` method is generated to write data in storage (Listing 5, line 5-7). To read data, `commandReceived()` method (Listing 5, line 12-16) is generated from the `ProfileDB` declaration in Vocabulary. The `profile` data access of `ProfileDB` maps badge identifier to a user's temperature preference ; in this case `commandReceived()` method is parametrized by the badge Identifier (Listing 5, line 9-11).

```
public abstract class ProfileDB {
  // HashMap for storing the association of  BadgeID and UserProfile
  private HashMap<String, UserProfile> Profile = new HashMap<String,
      UserProfile>();
  // This method is invoked by the application developer to store data in
      HashMap
  protected void setProfile(String newIndex, UserProfile newProfileValue) {
    Profile.put(newIndex, newProfileValue);
  }
  // This method get data from HashMap
  protected UserProfile getProfile(String index) {
    return Profile.get(index);
  }
  public final Object commandReceived(String methodName, Object arg) {
    if (methodName.equals("getProfile")) {
      return getProfile((String) arg);
    }
  }
}
```

Listing 5: The Java abstract class `ProfileDB` generated by framework generator from the declaration of the storage service `ProfileDB` in Vocabulary.

A controller uses the information generated by computational service to take the decisions that are carried out by triggering a command to actuator. To do so, concrete methods are generated to notify controller and to trigger actions of actuators. This is illustrate by the `RegulateTemp` controller. This component receives event notifications from other components using `notifiyReceived()` method (Listing 6, line 13-27). To trigger commands, `off()` and `setTemp()` methods (Listing 6, line 3-10) are generated by framework generator from the `RegulateTemp` declaration in architecture specification (Listing 2, line 25-26).

```
public abstract class RegulateTemp {

  protected void SetTemp(SetTempStruct arg) {
    //Trigger SetTemp comamnd with temperature Value
    PubSubMiddleware.sendCommand("SetTemp", arg);
  }
  protected void Off() {
    // Trigger Off Command.
    PubSubMiddleware.sendCommand("Off", null);
  }
```

```
11
12    // Received events from computational service.
13    public void notifyReceived(String eventName, Object arg,
14      Device deviceInfo) {
15      try {
16        if (eventName.equals("tempPref")) {
17
18          onNewtempPref((UserTempPrefStruct) arg);
19        }
20        if (eventName.equals("roomAvgTempMeasurement")) {
21
22          onNewroomAvgTempMeasurement((TempStruct) arg);
23        }
24
25      } catch (Exception e) {
26        e.printStackTrace();
27      }
28    }
29  }
```

Listing 6: The Java abstract class `RegulateTemp` generated by framework generator from the declaration of the controller `RegulateTemp` in Architecture.

An actuator has a set of actions that are triggered by controller. To do so, support is generated to allow actuator to receive commands from controller in form of concrete method. For example, from the `Heater` declaration in vocabulary specification (Listing 1, line 26-28), `Off()`, and `SetTemp()` methods are generated (Listing 7, line 3-5). Each method is then implemented in the subclass of `Heater`. To receive commands from controller, `commandReceived()` method (Listing 7, line 9-19) is generated.

```
1  public abstract class Heater{
2    // These method is implemented by application developer.
3    protected abstract void Off();
4
5    // This method is implemented by application developer.
6    protected abstract void SetTemp(SetTempStruct arg);
7
8    //This methods receives command from controller
9    public final Object commandReceived(String methodName, Object arg,
10       Device deviceInfo) {
11
12     if (methodName.equals("SetTemp") && arg instanceof TempStruct) {
13       SetTemp((TempStruct) arg);
14
15     } else if (methodName.equals("Off")) {
16       Off();
17     }
18     return null;
19   }
20 }
```

Listing 7: The Java abstract class `Heater` generated by framework generator from the declaration of the Heater entity in vocabulary.

Each entity is characterized by the type of information it generates and consumes. This information is managed by structure. For example, from the `struct` declaration of `BadgeDetectedStruct` in Vocabulary (Listing 2, line 7-9), variables, its constructor, and getter are generated (see Listing 8).

```
1  public class BadgeDetectedStruct {
2
3    // private variable of the structure
4    private String badgeID;
5
6    private String timeStamp;
7
```

```
  // Getters of member variable
  public String getbadgeID() {
    return badgeID;
  }

  public String getTimeStamp() {
    return badgeDetectedTimeStamp;
  }

  // This is constructor, which initialize BadgeDetected Struct
  public BadgeDetectedStruct(String badgeID, String timeStamp) {
    this.badgeID = badgeID;
    this.timeStamp = timeStamp;
  }
}
```

Listing 8: The Java abstract class `BadgeDetectedStruct` generated by framework generator from the Declaration of the `BadgeDetectedStruct` entity in Vocabulary

3.4 Target Network Specification

Given a vocabulary, Srijan Network Language (SNL) (for grammar, see Appendix A.3) is designed to enable the *network manager* to specify the details of each node in the system, including its placement (in terms of values of the region labels defined in the vocabulary), and abilities (a subset of those defined in the vocabulary). Much like SAL, the SNL for each application in a particular domain is customized using the vocabulary, and the abilities and region labels function as keywords. SNL is dedicated to describe targeted list of devices with region, and abilties. We present the SNL by examining network specification of smart office application (Listing 9, line 2-7). It specifies device with name `DOne`, its ability with `TemperatureSensor` and `BadgeReader`, and its region `building`, `floor`, and `room` with values 15, 11, and 0 respectively.

```
devices:
  DOne :
    region :
       Building : 15 ;
       Floor : 11;
       Room : 0;
    abilities :  TemperatureSensor ,  BadgeReader ;
```

Listing 9: Code snippets of smart office application network. Keywords from vocabulary are printed in word. Language keywords are printed in ***word***.

4 Evaluation

This section evaluates our approach and shows its ability to facilitate the development of the IoT applications. The goal of the evaluation is to demonstrate the advantage of our approach over manual application development approach. To achieve our objective, we explore *development time* aspects. We measure the quantity of the generated code in two applications developed by us: **smart office** (see Section 1.1 for description), and **fire management** application, which aims to detect fire in house and housing community (collection of houses). Fire is detected by analyzing data from smoke and temperature sensors. When the fire occurs, the application triggers sprinklers and alarms, and unlock doors to allows residents to evacuate the house and other residents of housing communities are informed by switch on lights. Table 2 gives number of each type of declaration in applications.

Application	Sensing Entity	Actuating Entities	Computational Service	Controller	Storage Service	Devices
Smart Office	12	6	13	6	2	8
Fire Management	8	8	4	3	0	8

Table 2: Number of declaration in our case study

Development time. It is defined as time required to develop an application. The more hand-written Lines of Code (LoC) there is, the time required to developing application is longer [24]. Our approach aims to

reduce the hand-written code and thus to reduce the development time. Our measures (using the Metrics 1.3.6[7] Eclipse plug-in) reveals that more than 81% of developed applications code is generated (see the Table 3), thus reducing the application development time. The measure of LoC is useful only if the generated code is executed. Otherwise generated code is large without affecting the development time. We measure the coverage of generated framework code (see the Table 4) during a number of executions of these applications, using the EclEmma[8] Eclipse plug-in. We studied that generated code and application code has high code-coverage. The part of the code that is not executed that most of them is error handling code or features that is not used by an application logic.

	Handwritten - Lines of Code				Generated - Lines of Code			
Application Name	Vocab Spec.	Arch. Spec.	Network Spec.	App. Logic	Partially generated App. Logic	Mapping code	Generated Framework	Generated Framework (in percentage)
Smart Office	30	36	49	169	139	470	638	81.45%
Fire Management	28	35	41	125	144	336	575	82.16%

Table 3: Lines of code in application development process

	Code Coverage	
Application	Application Logic	Generated Framework
Smart Office	95.8 %	90.7 %
Fire Management	95.4 %	93.3 %

Table 4: Code coverage of generated code

5 Related Work

In this section, we review existing approaches for the development of the Internet of Things applications. We illustrate each existing approach with representative examples.

Middleware. This approach provides programming support for acquiring and processing information from variety of sources. Numerous middleware have been proposed to support the implementation of applications [13, 15]. Olympus [25] is a programming framework on top of the Gaia middleware [26]. It allows developers to specify Active Space (enriched with sensors, actuators, users) as virtual entities. Virtual entities are described using high-level interface, allowing the developers to focus on the application logic. The framework takes care of resolving virtual entities into actual physical entities based on constraints, available resource in current space, and space-level policies (specified in high-level description). ContextToolkit [8] provides designers with the abstractions (context widgets, interpreters, aggregators, and services) on top of the distributed infrastructure [9].

Middleware hides the complexity of acquiring and processing sensor information. However, it often contains thousands of classes and its methods with many intricate dependencies that require significant technical background and considerable efforts to tune it properly.

Model-Driven Development (MDD). In this approach, software is developed not by directly writing code in the implementation languages, but it aims to specify the system using high-level abstract models (mainly expressed in UML) that can be transformed into code by automated code generators. PervML [27] allows developers to specify a pervasive systems at a high-level of abstraction by means of set of models (i.e., structural model, interaction model, functional model, services model, user model), which describes system functionality. From these models, system Java code is generated by the code generators. By following PervML approach, stakeholders get numerous benefits in application development such as rapid application development. However, this approach demands in-depth expertise of UML and its generic tools.

Currently, MDD is used to specify WSN applications at different-level of abstractions with the aim of code generation, energy consumption, and communication overhead. For example, in [20], the authors has presented a framework based on the Simulink, Stateflow, and Embedded Coder. In which, an engineer can specify sensor network components both at application-level and protocol-level. In [28, 29], the authors have presented MDD process to enable a low-cost prototyping and optimization of WSN is provided. The main focus of this work

[7] http://metrics.sourceforge.net/
[8] http://www.eclemma.org/

is on specifying set of modeling languages and transformation rules that transforms the models described by modeling languages to concrete one. The contribution of our approach is clear separation among vocabulary specification, software architecture specification of application, and deployment topology specification. This promotes separation of concerns among stakeholders, reuse of specification across projects and organizations.

Programming framework. This approach provides a language layer for describing the entities that are relevant to the application area, thus raising the level of abstraction. It assigns roles to stakeholders, providing separation of concerns. It includes compiler that take the design specifications written in their language as input and generate programming framework that support subsequent development stages. To the best our knowledge, there are very few programming frameworks have been proposed for IoT application development. ATaG and DiaSuite are representative examples of it.

ATaG [23] is a programming framework realing on a shared data pool for local communication. It helps developers to specify sense-and-react applications for WSAN. *Abstract task*, *abstract data item*, and *abstract channel* are core of the ATaG programming model. The abstract task is a logical entity encapsulating the processing of abstract data items, which represents the information. The flow of information among tasks is specified with abstract channels connecting a data item to the tasks. However, this framework is limited to WSAN applications where the device are often similar, thus heterogeneity remains unsolved.

DiaSpec [6] is a design language that allow developers to define a taxonomy dedicated to describing the class of entities (sensor, actuator, and storage) that are relevant to the target application, abstracting over their heterogeneity. This language also includes a layer to define the architecture of the application, following the sense-compute-control architecture pattern. However, this framework does not cover the large scalability issue of devices (i.e., multi-stage data processing, multiple-sub goals).

6 Discussion

Our work can be summarized with its key advantages as follows:

- **Identification of Roles in the Development Process.** We identify the precise role of each stakeholders involved in the development of IoT applications. This promotes separation of concerns and raises the level of knowledge that can be shared by other stakeholders. Thus, this clear identification of the expectations and specialized skills of each type of stakeholder smooths the IoT application development process.
- **A multi-stage model-based approach for IoT application development.** In order to support the different stages (i.e. design, implementation, and deployment) of the IoT application development process resulting from the actions of each of the stakeholders, we propose a multi-stage model-based approach. For each type of stakeholder, we provide a separate set of abstractions, customized not only for the IoT in general, but also for the application domain itself (*e.g.*, HVAC). This allows the stakeholder to specify their share at the *proper* level of abstraction commiserate with the skills and the responsibilities of the role he is playing.
- **Domain Specific Languages for stakeholders.** We note two key features of our languages. First, our proposed language focus on one particular aspect of application development, which significantly reduces the surface area that each roles need to learn. Second, it has very limited keywords to express specification at different stage, which makes it harder to say wrong things and easier to see when developers make errors.
- **A generative approach of application development.** We build an application development tool, combined with our language, provides support to various stakeholders to design and develop applications at each stages of IoT application development. The current version of our tool includes two code generators. First, *a*n application code generator produces skeleton classes from high-level specification (expressed using DSL at design stage). The generated code guides the development of application code and provides the application developers with high-level operations for component interactions. Second, *a* deployment code generator produces device-specific code to result in a distributed software system collaboratively hosted by individual devices in the IoT. We further show by our experiments that both code generators considerably reduces the development time.

7 Future Work

In our work so far, we have made initial progress towards providing support to all the stakeholders in the IoT application development process, and have prepared a foundation for our future work, which will proceed in two complementary directions, discussed below.

7.1 Supporting the Complete Set of IoT Application Characteristics

Our work so far, while extensive, does not fully cover all characteristics of IoT applications. Our immediate future research will review all the stages of the application development process, providing support for the following:

- **End-user interactions.** Modern smart phones are going to play a large part in the IoT, but they don't just contain sensors and actuators, but are also end-points of user interaction with the applications, which leads to hitherto-unseen traits. We will provide better ways for the stakeholders to define these software components at all stages of the application development process.
- **Complete support for storage services.** Our current work contains only a simplistic view of data storage services, which is inadequate given the rich diversity of data sources available today on the internet (*e.g.*, RDBMs and noSQL databases, using content that is both user generated such as photos as well as machine generated such as sensor data). Our work will provide abstractions to easily define the interfaces and interaction modes with the rich variety of data sources present in the IoT.
- **Novel mapping algorithms cognizant of heterogeneity.** While the notion of region labels is able to reasonably tackle the issue of scale at an abstraction level, the problem of heterogeneity among the devices still remains. We will provide rich abstractions to express both the properties of the devices (*e.g.,* processing and storage capacity, networks it is attached to, as well as monetary cost of hosting a software component), as well as the requirements from the stakeholders regarding the preferred placement of the software components of the applications. These will then be used to guide the design of algorithms for efficient mapping (and possibly migration) of software components on devices.

7.2 Comprehensive Evaluation of our Approach

The evaluation presented in Section 4 is preliminary. We plan to conduct an empirical evaluation based on a well-defined experimental methodology [14]. In particular, we explore two aspects: (1) *Productivity*, which evaluates development effort for developing application; and (2) *Evolution*, which evaluates time to correct, when there is a change in the user requirements. These are detailed below.

- **Productivity** In order to measure productivity, we plan to organize replicated-case study [3], which consist of suitable IoT application and two different teams (each consists of mainly three PhD students of our lab). These teams develop our application manually on top of the middle-ware layer and device-driver layer first, then using our approach. From this experiment, we measure *total time* (sum of learning time, development time, and deployment time) of each team in two approaches. The learning time reflects user intuitiveness of semantic model adopted by the DSL. The development and deployment time mainly propositional to lines of the written code.
- **Evolution** It is defined as time used to correct faults, when new entities are added, extended, or removed at any time and users have changing needs. In order to measure evolution aspect of our approach, we plan to select two different teams and a suitable case study with constant evolving requirements. The both teams develop our case study manually first, then using our approach. From this experiments, we measure *time to correct* of each team in two approaches.

8 Publications

1. P. Patel, A. Pathak, T. Teixeira, and V. Issarny. Towards application development for the internet of things. In Proceedings of the 8th IEEE ACM Middleware Doctoral Symposium, page 5. ACM, 2011.
2. P. Patel, A.Pathak, D. Cassou, and V.Issarny. Enabling High-Level Application Development in Internet of Things. (in preparation for submission to ICSE 2013).

References

[1] I. Akyildiz and I. Kasimoglu. Wireless sensor and actor networks: research challenges. *Ad hoc networks*, 2(4):351–367, 2004.

[2] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[3] V. Basili, R. Selby Jr, and D. Hutchens. Experimentation in software engineering. Technical report, DTIC Document, 1985.

[4] CASAGRAS EU project final report. `http://www.rfidglobal.eu/userfiles/documents/FinalReport.pdf`, 2009.

[5] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 431–440. IEEE, 2011.

[6] D. Cassou, J. Bruneau, C. Consel, and E. Balland. Towards a tool-based development methodology for pervasive computing applications. *Software Engineering, IEEE Transactions on*, (99):1–1, 2011.

[7] D. Cassou, J. Bruneau, J. Mercadal, Q. Enard, E. Balland, N. Loriant, and C. Consel. Towards a tool-based development methodology for sense/compute/control applications. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 247–248. ACM, 2010.

[8] A. Dey, G. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2-4):97–166, 2001.

[9] A. Dey, D. Salber, and G. Abowd. A context-based infrastructure for smart environments. 1999.

[10] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[11] A. Gluhak, M. Bauer, F. Montagut, V. Stirbu, M. Johansson, and M. Presser. Towards an architecture for the real world internet. *Towards the Future Internet*, page 313, 2009.

[12] S. Haller. The things in the internet of things. *Poster at the (IoT 2010). Tokyo, Japan, November*, 2010.

[13] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadis, M. Autili, M. A. Gerosa, and A. Ben Hamida. Service-Oriented Middleware for the Future Internet: State of the Art and Research Directions. *Journal of Internet Services and Applications*, 2(1):23–45, May 2011.

[14] B. Kitchenham, L. Pickard, and S. Pfleeger. Case studies for method and tool evaluation. *Software, IEEE*, 12(4):52–62, 1995.

[15] K. Kjær. A survey of context-aware middleware. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 148–155. ACTA Press, 2007.

[16] P. Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

[17] C. Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Prentice Hall PTR, 2004.

[18] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[19] L. Mottola and G. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):19, 2011.

[20] M. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, and S. Olivieri. A framework for modeling, simulation and automatic code generation of sensor network application. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON'08. 5th Annual IEEE Communications Society Conference on*, pages 515–522. Ieee, 2008.

[21] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498. ACM, 2007.

[22] P. Patel, A. Pathak, T. Teixeira, and V. Issarny. Towards application development for the internet of things. In *Proceedings of the 8th Middleware Doctoral Symposium*, page 5. ACM, 2011.

[23] A. Pathak, L. Mottola, A. Bakshi, V. Prasanna, and G. Picco. A compilation framework for macroprogramming networked sensors. *Distributed Computing in Sensor Systems*, pages 189–204, 2007.

[24] R. Picek and V. Strahonja. Model driven development-future or failure of software development. In *IIS*, volume 7, pages 407–413, 2007.

[25] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. Campbell, and M. Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*, pages 7–16. IEEE, 2005.

[26] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *Pervasive Computing, IEEE*, 1(4):74–83, 2002.

[27] E. Serral, P. Valderas, and V. Pelechano. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing*, 6(2):254–280, 2010.

[28] R. Shimizu, K. Tei, Y. Fukazawa, and S. Honiden. Case studies on the development of wireless sensor network applications using multiple abstraction levels. In *Software Engineering for Sensor Network Applications (SESENA), 2012 Third International Workshop on*, pages 22–28. IEEE, 2012.

[29] R. Shimizu, K. Tei, Y. Fukazawa, and S. Shinichi. Model driven development for rapid prototyping and optimization of wireless sensor network applications. In *Proceeding of the 2nd workshop on Software engineering for sensor network applications*, pages 31–36. ACM, 2011.

[30] R. Taylor, N. Medvidovic, and E. Dashofy. *Software architecture: foundations, theory, and practice*. Number 70. Wiley, 2009, Page -30.

[31] R. Taylor, N. Medvidovic, and E. Dashofy. *Software architecture: foundations, theory, and practice*. Number 70. Wiley, 2009, Page -565.

[32] R. Taylor, N. Medvidovic, and E. Dashofy. *Software architecture: foundations, theory, and practice*. Number 70. Wiley, 2009, Page -658.

[33] R. Want. An introduction to rfid technology. *Pervasive Computing, IEEE*, 5(1):25–33, 2006.

[34] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110. ACM, 2004.

9 Appendices

A DSL Grammars

A.1 SVL Grammar

```
VocSpec:

  /* Region Specification */
  ('regions' ':' (regions += Region)+ )+

  /*Structure Specification */
  ('structs' ':' (structs += Struct)* )*

   'abilities' ':'
  /* Sensor Specification */
  ('sensors' ':' (sensors += Sensor)+  )+

  /* Actuator Specification */
  ('actuators' ':' (actuators += Actuator)+ )+

  /* Storage Specification */
   ('storages' ':'  (storageService += StorageService)+ )*
;

/* Region Definition */
Region: regionLabel = RegionLabel ':' regionType = Type ';';
RegionLabel: name = ID ;

/* Structure Definition */
```

```
Struct: name = ID (fields += Field)+ ;
Field: name=ID ':' type += Type ';' ;

/* Sensor Definition */
Sensor:  sensorName = EntityName ((sources += Sources) )* ;
EntityName:  name = ID ;
Sources: 'generate' sourceName = SourceName ':' type = Type ';';
SourceName: name = ID;

/* Actuator Definition */
Actuator: actuatorName = EntityName ((actions += Actions))* ;
Actions: 'action' actionName= ActionName '(' (parameters += Parameters)* ')'
    ';' ;
ActionName: name = ID ;
Parameters: parameterName = ParameterName ':' type = Type ;
ParameterName:  name = ID ;



/*  Storage Definition  */

StorageService:
  storageServiceName = EntityName
  ((dataAccesses += DataAccess) )*
;

DataAccess:
  'generate' sourceName = SourceName ':' dataAccessType = Type
  'accessed-by' dataItem = ID ':' type = Type ';'
;

/* Type Definition */

Type :
  (primitiveDataType = PrimitiveDataType | structDef = [Struct])
;

PrimitiveDataType:
  INTEGER = 'Integer' | BOOLEAN = 'boolean' | STRING = 'String' | LONG = '
     long' | DOUBLE = 'double'
;
```

A.2 SAL Grammar

```
ArchSpec:

    /*Structure Specification */
   ('structs' ':' (structs += Struct)+ ) *

  'softwarecomponents' ':'

   /* Computational Service Specification*/
   ('computationalService' ':' (computationalService += ComputationalService
      )+ ) *

  /* Controller Specification */
   ('controller' ':' (controller += Controller)+ )+
;

/*  Computational Service Definition */
ComputationalService:
```

```
  computationalServiceName = ID
  ( (sources += Sources) |  // generate
  (inputs += Inputs) |  // Consume
  (requests += Requests )    |  // request to database service
  (deploymentAttribute += DeploymentAttribute))*
;

Requests:
  'request' requestname = [SourceName]  ';'
;

DeploymentAttribute:
  'in-region' ':' name=[RegionLabel] ';'
;

Inputs:
  'consume' name= [SourceName] ('from' 'hops' ':' regionID = INT ':'
     regionLabel = [RegionLabel] )? ';'
;

/*   Controller Service Definition */

Controller:
  controllerName = ID
  ((inputs += Inputs) |  // Consume
  (commands += Command)|
  (  deploymentAttribute += DeploymentAttribute)
  )*
;

Command:
  'command' commandName = [ActionName]  '(' (commandparameter =
     CommandParameter)?')'
  'to' 'hops' ':' regionID = INT ':'  regionLabel = [RegionLabel] ';'
;

CommandParameter:
  name = [ParameterName] ( ',' parameter = CommandParameter ) ?
;
```

A.3 SNL Grammar

```
NetworkSpec:
   ( 'devices' ':' (devices += Device)+ ) +
;

Device:
  deviceName =  ID ':'

  /* Device's region Specification */
  ( 'region' ':' ( deviceRegions += DeviceRegions)+ )*

  /* Device's Abilities  (i.e., Sensor, Actuator, Storage) */
  ( 'abilities' ':' (deviceAbilities += DeviceAbilities )+ )* ';'
;

DeviceRegions:
     regionLabel = [RegionLabel] ':' regionValue = INT ';'
;

DeviceAbilities :
```

```
  deviceEntityName = [EntityName] ( ',' name = DeviceAbilities ) ?
;
```

B Smart Office Application

B.1 Srijan Vocabulary Specification

```
regions:
   Building : Integer;
   Floor : Integer ;
   Room : Integer;

structs :
  BadgeDetectedStruct
    badgeID : String;
    timStamp : long;

  BadgeDisappearedStruct
    badgeID : String;
    timeStamp : long;

  TempStruct
    tempValue : double;
    unitOfMeasurement: String;

abilities:
  sensors:
     BadgeReader
       generate badgeDetected : BadgeDetectedStruct;
       generate badgeDisappeared :  BadgeDisappearedStruct;

     TemperatureSensor
       generate tempMeasurement : TempStruct;

  actuators:
     Heater
       action  Off();
       action SetTemp(setTemp :TempStruct);

     Monitor
       action Display(displayTemp : TempStruct);

  storages:
     ProfileDB
       generate  profile : TempStruct  accessed-by badgeID : String;
```

B.2 Srijan Architecture Specification

```
structs:
  UserTempPrefStruct
    tempValue : double;
    unitOfMeasurement : String;
    timeStamp : long;

softwarecomponents:
  computationalService:
    RoomAvgTemp
      generate roomAvgTempMeasurement : TempStruct;
      consume  tempMeasurement from hops: 0 :  Room ;
      in-region :  Room;

```

```
    FloorAvgTemp
      generate floorAvgTempMeasurement : TempStruct;
      consume roomAvgTempMeasurement from hops :0 : Floor;
      in-region : Floor;

    BuildingAvgTemp
      generate BuildingAvgTempMesurement : TempStruct;
      consume floorAvgTempMeasurement from hops : 0 : Building;
      in-region : Building;

    Proximity
      generate tempPref : UserTempPrefStruct;
      consume badgeDetected from hops : 0: Room;
      consume badgeDisappeared from hops : 0 : Room;
      request profile;
      in-region: Room;

    controller:
      RegulateTemp
        consume roomAvgTempMeasurement from hops : 0 : Room;
        consume tempPref from hops : 0 : Room;
        command Off() to hops : 0 : Room;
        command SetTemp(setTemp) to hops : 0 : Room;
        in-region : Room;

      ManageTemp
        consume BuildingAvgTempMesurement from hops:0: Building;
        command Display(displayTemp) to hops : 0: Building;
        in-region : Building;
```

B.3 Srijan Network Specification

```
devices:
  DOne :
    region :
       Building : 15 ;
       Floor : 11;
       Room : 0;
    abilities : TemperatureSensor , BadgeReader ;

    DTwo :
       region :
         Building : 15 ;
         Floor : 11 ;
         Room : 0 ;
       abilities : TemperatureSensor, Heater;

    DThree :
      region :
         Building : 15 ;
         Floor : 11 ;
         Room : 1 ;
      abilities : TemperatureSensor , Heater , ProfileDB , BadgeReader ;

    DFour :
      region :
         Building : 15;
         Floor : 11;
         Room : 1;
      abilities : TemperatureSensor ;
```

```
    DFive :
      region :
         Building : 15;
         Floor : 21 ;
         Room : 2;
      abilities : TemperatureSensor , BadgeReader;

    DSix :
       region :
         Building : 15;
         Floor : 21;
         Room : 2;
      abilities : TemperatureSensor, Monitor, Heater;

    DSeven :
      region :
         Building : 16;
         Floor : 14 ;
         Room : 3;
      abilities : TemperatureSensor , Monitor , BadgeReader;

    DEight :
      region:
         Building : 16;
         Floor : 14;
         Room : 3;
      abilities : ProfileDB, TemperatureSensor, Heater;
```

C Fire Management Application

C.1 Srijan Vocabulary Specification

```
regions :
  House : Integer;
  HousingCommunity : Integer;

structs :
  TempStruct
    tempValue : double;
    unitOfMeasurement : String;

  SmokePresenceStruct
    smokePresence : boolean;
    timStamp : long;

abilities:
  sensors:
    TemperatureSensor
      generate tempMeasurement : TempStruct;

    SmokeDetector
      generate smokPresence : SmokePresenceStruct;

    actuators:
       Door
        action Unlock();
       Alarm
        action Activate();
        action DeActivate();

       SprinklerSystem
```

```
        action Start();
        action Stop();

      Light
        action SwitchOn();
        action SwitchOff();
```

C.2 Srijan Architecture Specification

```
structs:
  HouseFireStruct
    smokePresence : boolean;
    timeStamp : long;
    tempValue : double;
    unitOfMeasurement : String;

softwarecomponents:
  computationalService:
    HouseAvgTempComputation
      generate houseAvgTemp : TempStruct;
      consume tempMeasurement from hops : 0: House;
      in-region : House;

    HouseFireComputation
      generate houseFireState : HouseFireStruct;
      consume houseAvgTemp from hops : 0 : House;
      consume smokPresence from hops : 0 : House;
      in-region : House;

    HcFireComputation
      generate hcFireState : HouseFireStruct;
      consume houseFireState from hops : 0 : HousingCommunity;
      in-region : HousingCommunity;

    controller:
      HouseFireController
        consume houseFireState from hops : 0 : House;
        command Activate() to hops : 0 : House;
        command DeActivate() to hops : 0 : House;
        command Start() to hops : 0: House;
        command Stop() to hops : 0 : House;
        command Unlock() to hops : 0 : House;
        in-region : House;

      HcFireController
        consume hcFireState from hops:0: HousingCommunity;
        command SwitchOn() to hops:0: HousingCommunity;
        command SwitchOff() to hops:0: HousingCommunity;
        in-region : HousingCommunity;
```

C.3 Srijan Network Specification

```
devices :
  DOne:
    region :
       HousingCommunity : 15;
       House : 101 ;
    abilities : TemperatureSensor , Door , Alarm;

  DTwo:
     region :
       HousingCommunity : 15;
```

```
11        House : 101;
12      abilities:  TemperatureSensor , SmokeDetector ;
13
14    DThree :
15      region:
16        HousingCommunity : 15 ;
17        House : 101 ;
18      abilities :  TemperatureSensor ,  SprinklerSystem;
19
20    DFour :
21      region :
22        HousingCommunity : 15 ;
23        House : 101;
24      abilities :  Light;
25
26    DFive:
27      region :
28        HousingCommunity : 15 ;
29        House : 102;
30      abilities :   TemperatureSensor ,  SmokeDetector , Alarm;
31
32    DSix :
33       region :
34        HousingCommunity : 15 ;
35        House : 102;
36      abilities :  TemperatureSensor ,  SprinklerSystem;
37
38    DSeven :
39      region :
40        HousingCommunity : 15;
41        House : 102;
42      abilities :  TemperatureSensor ,  Door ;
43
44    DEight :
45      region :
46         HousingCommunity : 15;
47         House : 102 ;
48      abilities :  Light;
```